# COL106: Data Structures, I Semester 2015-16

Assignment 4

A small search engine

September 26, 2015

In this assignment we will build the basic data structure underlying search engines: an *inverted index*. We will use this inverted index to answer some simple search queries.

## An inverted index for a set of webpages

Suppose we are given a set of webpages $W$. For our purposes, each webpage $w \in W$ will be considered to be a sequence of words $w_1, w_2, \ldots w_k$. Another way of representing the webpage could be to maintain a list of words along with the position(s) of the words in the webpage. For example consider a web page with the following text:

Data structures is the study of structures for storing data.

This can be represented as

$$\{(\text{data} : 1, 10), (\text{structures} : 2, 7), (\text{study} : 5), (\text{storing} : 9)\}.$$

Note that the small connector words like "is", "the", "of", "for" have not been stored. Words like this are referred to as *stop words* and are generally removed since they are very frequent and normally contain no information about the content of the webpage.

This representation of the webpage is similar to the index we see at the back of many books which tell us the page numbers where certain important terms used in the book may be found. In fact, we can refer to this as an

*index* for the webpage. In mathematical notation we would say that given a webpage $w = w_1, w_2, \ldots, w_k$, the index of $w$ is

$$\{(u : i_1(u), \ldots i_\ell(u)) : w_{i_j(u)} = u, 1 \leq j \leq \ell\}.$$

An index is used to find the location of a particular string (word) in a specific document or webpage, but when we move to a *collection* of webpages, we need to first figure out which of the web pages contain the string. For this we store an *inverted index*. Let us try to define an inverted index formally.

Let us suppose we are given a collection $\mathcal{C}$ of webpages. For each page $p \in C$, let us denote by $W(p)$ the set of all words (excluding stop words) that occur in $p$. Note that

$$W(\mathcal{C}) = \bigcup_{p \in \mathcal{C}} W(p),$$

is the set of all words in our collection.

An inverted index for $\mathcal{C}$ will contain an entry for each word $w \in W(\mathcal{C})$. This entry will contain tuples of the form $(p, k)$ to indicate that $w$ occurs in the $k$th position of page $p \in \mathcal{C}$. Using the notation that $p[k]$ denotes the $k$th word of page $p$, we can say that the inverted index of $\mathcal{C}$ is defined as

$$\mathrm{Inv}(\mathcal{C}) = \{(w : \{(p, k) : p \in \mathcal{C}, p[k] = w\}) : w \in W(\mathcal{C})\}.$$

For example, consider the following (small) collection of documents.

1: Data structures is the study of structures for storing data.

2: Structural engineers collect data about structures

The inverted index for this would be

```
{(data : {(1,1), (1,10), (2,4)}),
(structures: {(1,2), (1,7), (2,6)}),
(study : {(1,5)}),
(storing : {(1,9)}),
(structural : {(2,1)}),
(engineers : {(2,2)}),
(collect : {(2,3)}) }
```

## The web search problem

The *web search problem* is defined as follows:

> Given a collection of webpages $\mathcal{C}$ and a sequence of words $q_1 \dots q_k$, find the "most relevant" set of pages $p_1, p_2, \dots p_\ell$ that contain as many of $q_1 \dots q_k$ as possible and return them in the order of decreasing "relevance."

The question of how to measure the relevance of a webpage to a particular query is an involved question with no easy answers. However, for the purpose of this assignment we will work with a simple scoring function.

### A scoring function for search term relevance

Given a word $w$ and a webpage $p$, if $w$ occurs $\ell$ times in positions $k_1, \dots, k_\ell$, the relevance score of the page $p$ for the word $w$ is defined as

$$\text{relevance}_w(p) = \sum_{i=1}^{\ell} \frac{1}{k_i^2}.$$

So, if we are given a search query that has a single term, say $w$, to return the webpages in order of relevance we have to first extract the entry corresponding to $w$ from $\text{Inv}(\mathcal{C})$ and then calculate the relevance of each page and return the pages in decreasing order of relevance.

### Compound searches

In this assignment we will answer three kinds of search queries: AND queries, OR queries and phrase queries. We now describe these three along with their scoring methodology.

- **OR** queries: Given a search query $q_1 \dots q_k$, any page that contains *any* of the words $q_1$ to $q_k$ is a valid answer. The relevance score of a page $p$ is computed as

  $$\text{relevance}_{q_1 \dots q_k}(p) = \sum_{i=1}^{k} \text{relevance}_{q_i}(p),$$

  and pages are returned in decreasing order of relevance. Note that if some $q_i$ does *not* occur in page $p$ the $\text{relevance}_{q_i}(p) = 0$.

- **AND** queries: Given a search query $q_1 \dots q_k$, any page that contains *all* of the words $q_1$ to $q_k$ is a valid answer. The relevance score of a page $p$ is computed as

$$\text{relevance}_{q_1 \dots q_k}(p) = \sum_{i=1}^{k} \text{relevance}_{q_i}(p),$$

  and pages are returned in decreasing order of relevance.

- **Phrase** queries : Given a search query $q_1 \dots q_k$, any page that contains $q_1$ in position $\ell$, $q_2$ in position $\ell+1$ and so on till $q_k$ in position $\ell+k-1$ is said to contain the phrase $q_1 \dots q_k$ at the position $\ell$. Suppose a webpage $p$ contains the phrase $q_1 \dots q_k$ at positions $\ell_1, \ell_2, \dots \ell_m$ then the relevance score of page $p$ for this phrase is

$$\text{relevance}_{q_1 \dots q_k}(p) = \sum_{i=1}^{m} \frac{1}{\ell_i^2}.$$

The implementation of the search engine has been divided in to two assignments. In the first assignment, Assignment 4, we will build the basic backbone of the search engine and answer single word search queries. Compound queries will be implemented in Assignment 5.

## Assignment 4
## Deadline: **11:55 PM, 4 October 2015**

- Write a Java class `MySet` using Java generic's (*https://docs.oracle.com/javase/tutorial/java/generics/types.html*). The class should be represented as *MySet<X>* where X is the datatype of the set. MySet should implement the following methods:
  - `void addElement(X element)`: Add `element` to the set.
  - `MySet<X> union(MySet<X> otherSet)`: Return MySet which represents a union of the current set and the `otherSet`.
  - `MySet<X> intersection(MySet<X> otherSet)`: Return MySet which represents an intersection of the current set and the `otherSet`.

- Write a Java class `MyLinkedList` using Java generic's. It should contain the standard methods of a linked list.
- Write a Java class `Position` that represents a tuple <*page p, word position i*>.
  - `Position(PageEntry p, int wordIndex)` Constructor method.
  - `PageEntry getPageEntry()` Return p
  - `int getWordIndex()` Return wordIndex
- Write a Java class `WordEntry`. For a string *str*, this class stores the list of word indice's where str is present in the document(s).
  - `WordEntry(String word)`: Constructor method. The argument is the word for which we are creating the word entry.
  - `void addPosition(Position position)`: Add a position entry for str.
  - `void addPositions(MyLinkedList<Position> positions)`: Add multiple position entries for str.
  - `MyLinkedList<Position> getAllPositionsForThisWord()`: Return a linked list of all position entries for str.
- Write a Java class `PageIndex` which stores one word-entry for each *unique* word in the document.
  - `void addPositionForWord(String str, Position p)`: Add position `p` to the word-entry of `str`. If a word entry for `str` is already present in the page index, then add p to the word entry. Otherwise, create a new word-entry for `str` with just one position entry `p`.
  - `LinkedList<WordEntry> getWordEntries()`: Return a list of all word entries stored in the page index.
- Write a Java class `PageEntry` to store the the information related to a webpage. It should contain following methods:
  - `PageEntry(String pageName)`: Constructor method. The argument is the name of the document. Read this file, and create the page index.

  - `PageIndex getPageIndex()`: This method returns the page index of this web-page.

- Write a Java class `MyHashTable` that implements the hashtable used by the InvertedPageIndex. It maps a word to its word-entry.
  - `private int getHashIndex(String str)`: Create a hash function which maps a string to the index of its word-entry in the hashtable. The implementation of hashtable should support chaining.
  - `void addPositionsForWord(WordEntry w)`: This adds an entry to the hashtable: *stringName(w)* $->$ *positionList(w)*. If no word-entry exists, then create a new word entry. However, if a word-entry exists, then merge `w` with the existing word-entry.

- Write a Java class `InvertedPageIndex` which contains the following methods:
  - `void addPage(PageEntry p)`: Add a new page entry `p` to the inverted page index.
  - `MySet<PageEntry> getPagesWhichContainWord(String str)`: Return a set of page-entries of webpages which contain the word `str`.

- Write a Java class `SearchEngine`. This is the class that we will use as an interface to the search engine. It should contain following methods:
  - `SearchEngine()`: This is the constructor method. It should create an empty InvertedPageIndex.
  - `void performAction(String actionMessage)`: This the main stub method that you have to implement. It takes an action as a string. The list of actions, and their format will be described later.

Actions:

- `addPage x` Add webpage `x` to the search engine database. The contents of the webpage are stored in a file named `x` in the `webpages` folder.

- `queryFindPagesWhichContainWord x` Print the name of the webpages which contain the word `x`. The list of webpage names should be comma separated. If the word is not found in any webpage, then print "No webpage contains word `x`"

- `queryFindPositionsOfWordInAPage x y` Print the word indice's where the word `x` is found in the document `y`. The word indices should be separated by a comma. If the word `x` is not found in webpage `y`, then print "Webpage `y` does not contain word `x`".

Points to note:

- Convert each word to lowercase.

- Do not store the connector words in the search engine. However, consider them when you calculate the word indice's. Here is a list of connector words: { *a, an, the, they, these, for, is, are, of, or, and, does, will, whose* }.

- Ignore the punctuation marks completely i.e. your search engine should give the same results even if we remove the punctuation marks from the document. Here is a list of punctuations: {*. , ; ' " ? # ! - :*}
  So, these words mean the same thing: (stacks, stack's, "stacks", 'stacks')

- *Plural and singular form*: Assume that these words are same: (stack, stacks), (structure, structures).